MTR110043
MITRE TECHNICAL REPORT

A Comparative Study of PDF Generation Methods:

Measuring Loss of Fidelity When Converting Arabic and Persian MS Word Files to PDF

Contract No.: W15P7T-11-C-F600
Project No.: 0711G01Z-HB

Paul M. Herceg
Catherine N. Ball

February 17, 2011

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **17 FEB 2011**	2. REPORT TYPE	3. DATES COVERED **00-00-2011 to 00-00-2011**
4. TITLE AND SUBTITLE **A Comparative Study of PDF Generation Methods: Measuring Loss of Fidelity When Converting Arabic and Persian MS Word Files to PDF**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **MITRE Corporation,7525 Colshire Drive,McLean,VA,22102**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		

14. ABSTRACT

Converting files to Portable Document Format (PDF) is popular due to the format?s many advantages. For example, PDF allows an author to control or preserve the rendering of a digital document, distribute it to other systems, and ensure that it displays in a viewer as intended. From the perspective of Human Language Technology (HLT), however, PDFs are problematic. PDF is a display-oriented digital document format; the point of PDF is to preserve the appearance of a document, not to preserve the original electronic text. We observed errors in PDF-extracted text indicating that either the PDF generator or extractor, or both, mishandled the document structure, character data, and/or entire textual objects. And we learned that other HLT researchers reported data loss when extracting electronic text from PDFs. This motivated further study of digital document data exchange using PDFs. MITRE conducted an exploratory study of data exchange using PDF in order to investigate the data loss phenomenon. We limited our study to Middle Eastern electronic text: specifically Arabic and Persian. The study included a test for scoring PDF generation methods?(a) using a common, best-practice setup to generate PDFs and extract text, and (b) using character accuracy to quantify the quality of PDF-extracted text. We ranked 8 methods according to the resulting accuracy scores. The 8 methods map to 3 core PDF generation classes. At best, the Microsoft Word class resulted in 42% Overall Accuracy. Best scores for the PDFMaker and Acrobat Distiller/PScript5.dll classes were 95% and 96%, respectively. This paper explains our tests and discusses the results, including evidence that using PDF for data exchange of typical Arabic and Persian documents results in a loss of important electronic text content. This loss confuses human language technologies such as search engines, machine translation engines, computer-assisted translation tools, named entity recognizers, and information extractors. Furthermore, most of the spurious newlines, spurious spaces in tokens, spurious character substitutions, and entity errors observed in the study were due to the PDF generation method rather than the PDF text extractor. So, using a common configuration to convert reliable electronic text to PDF for data exchange causes irretrievable loss of electronic text on the receiving end.

15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**24**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Abstract

Converting files to Portable Document Format (PDF) is popular due to the format's many advantages. For example, PDF allows an author to control or preserve the rendering of a digital document, distribute it to other systems, and ensure that it displays in a viewer as intended.

From the perspective of Human Language Technology (HLT), however, PDFs are problematic. PDF is a display-oriented digital document format; the point of PDF is to preserve the *appearance* of a document, not to preserve the original electronic text. We observed errors in PDF-extracted text indicating that either the PDF generator or extractor, or both, mishandled the document structure, character data, and/or entire textual objects. And we learned that other HLT researchers reported data loss when extracting electronic text from PDFs. This motivated further study of digital document data exchange using PDFs.

MITRE conducted an exploratory study of data exchange using PDF in order to investigate the data loss phenomenon. We limited our study to Middle Eastern electronic text: specifically Arabic and Persian. The study included a test for scoring PDF generation methods—(a) using a common, best-practice setup to generate PDFs and extract text, and (b) using character accuracy to quantify the quality of PDF-extracted text. We ranked 8 methods according to the resulting accuracy scores. The 8 methods map to 3 core PDF generation classes. At best, the Microsoft Word class resulted in 42% Overall Accuracy. Best scores for the PDFMaker and Acrobat Distiller/PScript5.dll classes were 95% and 96%, respectively.

This paper explains our tests and discusses the results, including evidence that using PDF for data exchange of typical Arabic and Persian documents results in a loss of important electronic text content. This loss confuses human language technologies such as search engines, machine translation engines, computer-assisted translation tools, named entity recognizers, and information extractors.

Furthermore, most of the spurious newlines, spurious spaces in tokens, spurious character substitutions, and entity errors observed in the study were due to the PDF generation method, rather than the PDF text extractor. So, using a common configuration to convert reliable electronic text to PDF for data exchange causes irretrievable loss of electronic text on the receiving end.

Keywords: Digital Documents, File Conversion, Reliable Electronic Text, Human Language Technology, Portable Document Format, PDF, Microsoft Word, DOCX, Arabic, Persian, Character Error Rate, Data Exchange

Table of Contents

1. Introduction

Converting files to PDF is a popular practice for exchanging digital documents. Converting a document to PDF offers many advantages. The PDF can be viewed using the widely available Adobe Reader or other readers, without any need for the application that created the original file; the PDF can preserve the look and feel of the original document; and the PDF can preserve the integrity of the original document, in the sense that a PDF file is less easily modified than (say) a Microsoft Word document.

From the perspective of Human Language Technology (HLT), however, exchanging digital documents via PDF is highly problematic. Herceg and Ball (2010) explain that HLT applications such as machine translation and information extraction require *reliable electronic text* as input, and that extracting text from a file is one of the first steps in a document processing pipeline. Extracting reliable electronic text from PDFs is fraught with difficulties, particularly for foreign script languages (Herceg & Ball, 2010, p. 3). Issues arise because PDF is a display-oriented document format; the point of PDF is to preserve the *appearance* of a document, not to preserve the original electronic text.

The PDF file format is designed to provide instructions for painting (or rendering) document objects on one or more virtual document pages (Adobe, 2008, p. vii; King, 2008; Powley et al., 2009, p. 3). The process of converting to PDF Language text rendering instructions must be reversed to derive the original source electronic text.

The complexity of reversing this conversion makes extracting the electronic text challenging. The text rendering instructions use strings of character codes, denoting glyphs (in a font encoding), from which electronic text can be derived only under certain conditions (Adobe, pp. 251, 292).[1] King (2008) and Carrier (2009) explain the following:

- A glyph's character code may be the code point value in a standard encoding such as ASCII or Unicode, but this is not always the case—for example, the character codes for ligatures in a Latin font, and the character codes for glyphs in an Arabic font
- Extracting text from a PDF requires decoding strings of character codes via a character mapping table (which may or may not be present), reforming words (e.g., words that were broken into two or more chunks, or that were hyphenated and broken across two lines), and reconstituting newlines and reading order based on spatial or structural information
- Furthermore, extracting text from a PDF requires normalizing decomposable Unicode characters, and, converting right-to-left language from *display order* into *storage order*, while not affecting left-to-right components

Powley, Dale, and Anisimoff (2009) experienced these complexities firsthand while trying to perform information extraction research on PDFs of English-language academic papers (p. 3). They mention tests of a number of commercial and open source text extractors. None were able to generate *reliable electronic text* acceptable for their research.

[1] A *glyph* is a particular shape of a given letter or character.

We, also, have observed errors in PDF-extracted text indicating that either the PDF generator or text extractor, or both, mishandled the document structure, character data, and/or entire textual objects. We are not aware of any research that has sufficiently measured this phenomenon.

PDFs may be generated by a variety of methods, such as printing to PDF, converting a document to PDF using Adobe Acrobat Professional, converting a document to PDF using Microsoft Word, and so on. We believe that the amount of mishandling introduced by each of these methods varies, and that it would be valuable to rank these techniques in order of accuracy (e.g., best accuracy at the top of the list). Furthermore, we believe that such data loss may happen more frequently with non-western languages.

MITRE undertook to rank a number of PDF generation methods by scoring the accuracy of PDF text extraction. For the study, we used a test pipeline from plain text, to Microsoft Word format, to PDF, to PDF-extracted plain text—where we held all components constant and systematically manipulated the document language, the PDF generation method, and the Unicode normalization treatment. We sought to design the test to be economical. Hence, we limited our focus to (a) Arabic and Persian; (b) methods that present themselves on a PC with Microsoft Windows XP, Microsoft Word, and Adobe Acrobat; (c) a small test data set; and (d) using off-the-shelf tools to calculate accuracy.

2. Method

In this section, we discuss the details of the procedure we used to achieve the ranking of PDF generation methods. Figure 1 shows a notional view of the test—a document processing pipeline. It begins with an original plain text document in Unicode UTF-8 encoding. Microsoft Word converts this document to its proprietary .DOCX format. One of a number of PDF generation methods converts the file to PDF. A PDF text extractor delivers the content as a PDF-extracted plain text file (e.g., UTF-8). Finally, a scoring tool automatically scores the character accuracy of the PDF-extracted document against the original document.

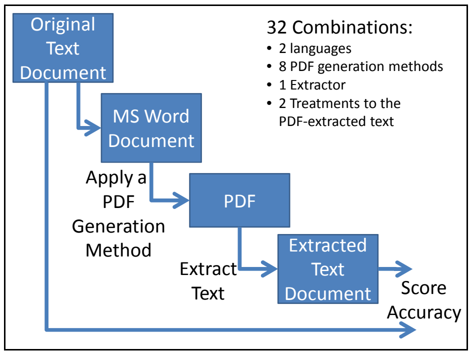

Figure 1. Notional view of the test.

We ran the test 32 times, introducing a slight variation in the configuration of components with each run. Variations included two Middle Eastern languages, eight PDF generation methods, and two treatments to the extracted text. We used the same text extractor for all runs.

The original plain text documents we used in our test are not real-world documents. Rather, they are space-delimited lists of tokens (i.e., words and word phrases), symbols, and entities that were carefully fabricated for the purpose of the test. Before we discuss the process the team used to develop these text documents in each language, we need to convey background about electronic text in the PDF file format.

2.1. Snippets of Electronic Text Reside in PDF Language Commands as Font Codes

Simply put, electronic text in a PDF file resides in PDF Language statements (or commands) that specify sequences of *font codes* as arguments (Adobe, 2008, p. 251).[2,3] We use the non-standard term *font code* for the purpose of simplifying the discussion of PDFs. These *font codes* are the lookup values for a font encoding table, and denote a specific *glyph* in a font such as Arial or Times New Roman.

Sequences, or *strings*, of font codes are encoded snippets of electronic text that are limited to the length of the widest line of text on a given page. For the Arabic and Persian PDFs we have examined, each font code is a 4-byte sequence; hence, font code strings are not readable Arabic and Persian. To be clear, Arabic and Persian electronic text in a PDF does *not* appear as a stream of characters in a widely known encoding (e.g., Unicode, CP1256). To be accurate, however, the exception is where a string's font codes represent Latin glyphs (or other glyphs appearing in the ASCII character set). In these Latin strings, each font code is a single byte that is the same as the ASCII code point for lookup in an ASCII table. These Latin font code strings can be copied from their PDF Language context and used as ASCII strings.

A PDF contains a list of PDF Language commands for rendering the strings of font codes (i.e., textual glyphs), and/or other document objects, on one or more electronic document pages. These commands have their roots in typography, so, to understand the PDF Language, one must have at least a rudimentary understanding of typography. Only a few of the typographic commands, called *text showing operators*, can hold strings of font codes as arguments. How these operators behave when rendering each textual glyph on a Cartesian coordinate system is based on the typographic settings configured or re-configured with a larger set of typographic commands: text state operators, text positioning operators, and text object operators. The PDF Language is cryptic to the untrained eye. To get a sense of its complexity: a PDF may contain several hundred typographic commands to render a simple paragraph of 100 words.

Often the typographic commands for painting strings of font codes (i.e., strings of electronic text) appear out of their natural reading order in the PDF Language code. Also, each entire line of text may be dispersed among several text showing operators, for which each font code string is a single token/word, part of a token/word, or merely a single character of a token/word. Because glyphs are rendered on a page's coordinate system, there really is no requirement that the order of the text showing operators reflect reading order (also known as *text flow*). The degree to which electronic text is dispersed among multiple PDF commands depends on the method responsible for generating the PDF.

[2] *Font code* is short for *font character code*.

[3] Technically speaking, the PDF specification uses the term *character code* rather than *font code*. However, using the term *character code* in this paper would make it difficult for readers to differentiate between (a) character codes that are font encoding values and (b) character codes that are code points in a widely known standard encoding (e.g., Unicode). Furthermore, using the term *character code* here would introduce undue complexity because the PDF specification actually gives three definitions of the term *character code*, and states that the meaning varies depending on context (Adobe, 2008, p. 6).

For Arabic and Persian, there is yet another nuance to the font code strings. In the Arabic and Persian PDF files we examined, each string of font codes is stored in reverse reading order—contrary to the way Unicode text is stored.

2.2. Arabic and Persian Electronic Text is Coded in Contextual Forms in a PDF

Arabic and Persian electronic text in a PDF is coded in its contextual forms. For Arabic and Persian, each letter of the alphabet can take on a number of contextual forms (i.e., different shapes)—isolated, initial, medial, and final. As we already mentioned, each code in a string of font codes indicates a particular glyph shape, which is looked up in a font encoding table (e.g., Arial, Times New Roman). For most of the letters of the Arabic and Persian alphabets, each form is associated with a different glyph shape in a given font encoding table. An Arabic or Persian font must at least support the basic contextual shapes, but it may include even more shapes. The combination of a character with a diacritic may add yet another glyph shape. The special additional characters in Arabic and Persian may add many more glyph shapes.

2.3. Test Documents

With this background on electronic text in PDFs, and, in particular, Arabic and Persian, we turn to the process that our team used to develop the Arabic and Persian test documents. Our test documents take into account all of the contextual forms, the character and diacritic combinations, and the special characters used most widely in each of the languages. Also, the documents, although fabricated, incorporate words in which these contextual forms occur.

We performed the following steps for each language. We developed a list of the canonical Unicode code points typically used in the given language (Table 1 and Table 2). Then, we applied an understanding of the language to expand this list to the various glyph shapes typically rendered by each code point.[4,5] Google's language-specific search allowed us to download Microsoft Word documents in the language. Detailed searching within these files allowed us to generate a list of tokens representing all of the glyph shapes—one token for each glyph shape.[6,7]

[4] Arabic numerals used in the West were not included.
[5] Some code points were added later. We added ARABIC TATWEEL (ـ U+0640) and ARABIC FATHATAN (◌ً U+064B) to the Arabic and Persian lists. And we added ARABIC SHADDA (◌ّ U+0651) to the Persian list.
[6] Most of the tokens are single words, while others are more than one word. We could achieve the study's objective regardless of the number of words per token.
[7] At this point in the process we had a total of 127 code point and contextual form combinations for Arabic, and 146 combinations for Persian. These counts do not include the ZERO WIDTH NON-JOINER.

Table 1. Unicode Code Points for Developing the Arabic Test Document

- All canonical code points for the 28 letters of the alphabet
- ARABIC LETTER HAMZA (ء U+0621) and the following alphabetic characters:
  ARABIC LETTER ALEF WITH HAMZA BELOW (إ U+0625),
  ARABIC LETTER ALEF WITH HAMZA ABOVE (أ U+0623),
  ARABIC LETTER WAW WITH HAMZA ABOVE (ؤ U+0624),
  ARABIC LETTER YEH WITH HAMZA ABOVE (ئ U+0626)
- The following modified letters:
  ARABIC LETTER ALEF WITH MADDA ABOVE (آ U+0622),
  ARABIC LETTER TEH MARBUTA (ة U+0629),
  ARABIC LETTER ALEF MAKSURA (ى U+0649)
- The ZERO WIDTH NON-JOINER (U+200C)
- Arabic-Indic Digits ٠ (U+0660), ١ (U+0661), ٢ (U+0662), ٣ (U+0663), ٤ (U+0664), ٥ (U+0665), ٦ (U+0666), ٧ (U+0667), ٨ (U+0668), ٩ (U+0669)

Table 2. Unicode Code Points for Developing the Persian Test Document

- All canonical code points for the 32 letters of the alphabet, including the following four letters unique to the Persian alphabet:
  ARABIC LETTER PEH (پ U+067E),
  ARABIC LETTER TCHEH (چ U+0686),
  ARABIC LETTER JEH (ژ U+0698),
  ARABIC LETTER GAF (گ U+06AF)
- ARABIC LETTER HAMZA (ء U+0621) and the following alphabetic characters:
  ARABIC LETTER ALEF WITH HAMZA ABOVE (أ U+0623),
  ARABIC LETTER WAW WITH HAMZA ABOVE (ؤ U+0624),
  ARABIC LETTER YEH WITH HAMZA ABOVE (ئ U+0626),
  ARABIC LETTER HEH WITH YEH ABOVE (ۀ U+06C0)
- The commonly used Arabic substitutions for Persian characters:
  ARABIC LETTER YEH (ي U+064A),
  ARABIC LETTER KAF (ك U+0643),
  ARABIC LETTER ALEF MAKSURA (ى U+0649)
- The ZERO WIDTH NON-JOINER (U+200C)
- Arabic-Indic Digits ٠ (U+0660), ١ (U+0661), ٢ (U+0662), ٣ (U+0663), ٤ (U+0664), ٥ (U+0665), ٦ (U+0666), ٧ (U+0667), ٨ (U+0668), ٩ (U+0669)
- Eastern Arabic-Indic Digits ۰ (U+06F0), ۱ (U+06F1), ۲ (U+06F2), ۳ (U+06F3), ۴ (U+06F4), ۵ (U+06F5), ۶ (U+06F6), ۷ (U+06F7), ۸ (U+06F8), ۹ (U+06F9)

According to guidance from the Unicode Consortium, this list of Unicode code points includes only those from the canonical range, with the exception of the ZERO WIDTH NON-JOINER. It does not include Unicode presentation forms.[8] The Unicode Consortium (2011) explains:

- Data files should only include the canonical range
- Presentation forms merely exist for historical reasons
- It is the responsibility of software using plain text data to render applicable Arabic ligatures and contextual forms

[8] Specifically, the presentation forms are Arabic Presentation Forms A (U+FB50 to U+FDFF) and Arabic Presentation Forms B (U+FE70 to U+FEFF).

We considered each digit to be a token and added it to the list. We also added Allah (ﷲ , U+0627 U+0644 U+0644 U+0647) and a short list of date and measurement entities.[9]

We concatenated all of the tokens on the list, delimiting each with a space, and exported the document (Microsoft Word format) to plain text (UTF-8). As a final check, we validated that conversions to Microsoft Word format, and back again to UTF-8, generated an identical plain text UTF-8 file.

Following the process above, we generated a single UTF-8 test document for Arabic and a single UTF-8 test document for Persian.

2.4. Conversion to Microsoft Word Format

With the original plain text UTF-8 files in hand, we initiated the document processing pipeline. For each original plain text file, we performed the following:

- Opened the UTF-8 file in Microsoft Word 2007 SP2
- Changed the font to 16-point Arial
- On the Home Tab, in the Paragraph Group, clicked the Right-to-Left Text direction button
- Saved the .DOCX file (i.e., Microsoft Word file)

2.5. PDF Generation

Next, we converted each .DOCX file to PDF. We constrained the test to the 8 PDF generation methods that can be performed on a PC with Windows XP SP3, Microsoft Word 2007 SP2, and Adobe Acrobat 9 Professional (v.9.4.1). Correspondence with Adobe Systems confirmed that these 8 compose an exhaustive list of such PDF generation methods:

1. Open the .DOCX file in Acrobat; click File > Save As; click Save (to PDF)
2. Open Acrobat; click File > Create PDF > From File; specify the .DOCX file; click Open; click Save (to PDF)
3. Open the .DOCX file in Word; print it to the Adobe printer (i.e., virtual printer)
4. Open the .DOCX file in Word; click the Microsoft Office Button; click Save As > Adobe PDF
5. Open the .DOCX file in Word; click the Microsoft Office Button; click Save As > PDF or XPS; click Publish
6. Open the .DOCX file in Word; click the Acrobat tab; click Create PDF
7. Ensure the default printer is set to the Adobe printer (i.e., virtual printer); right-click the .DOCX file in Windows Explorer; click Print
8. Right-click the .DOCX file in Windows Explorer; click Convert To Adobe PDF; click Save

Before applying these methods, we prepared Adobe Acrobat preferences. We ensured that the Adobe Printer “Printing Preferences” were set to default, and we set our default printer to the Adobe PDF virtual printer. In Microsoft Word, we ensured the Adobe Acrobat PDFMaker

[9] Specifically, we added 6 entities to the Arabic list and 5 entities to the Persian list.

preferences were set to defaults, except for the following setting: on the PDFMaker settings tab, we checked *Create PDF/A-1a:2005 compliant file*. This setting is intended to ensure that the structure and semantics of the original digital document are preserved. We did this to give the PDFMaker-related PDF generation methods the best possible chance of preserving the electronic text of our documents.

2.6. PDF Text Extraction

We held the PDF text extractor constant throughout the study, and we ensured that we used a best-of-breed extractor. Several text extractors are available, but performance varies widely (Powley, et al., 2009, p. 3). We used PDF Box because it includes a best-of-breed extractor that can be invoked from the command line (Apache, 2010).

Specifically, we used the following command line:

```
	java -Dglyphlist_ext="glyphlist-ext.txt" -jar "c:\progra~1\pdfbox\pdfbox-app-
	1.3.1.jar" ExtractText -sort -encoding UTF-8 output.txt
```

This command line specifies a glyph list file. Although a glyph list file is available directly from Adobe Systems, we used a glyph list file supplied from a collaborating group that optimized the glyph list for Arabic and Persian. At the time of this writing, the Adobe list is available for download at http://partners.adobe.com/public/developer/en/opentype/glyphlist.txt

2.7. Scoring

The goal of the study was to rank PDF generation methods based on how well a PDF-extracted text document compares to its original plain text document. Our past collaboration with commercial content extraction vendors indicated that there are no standard measures or practices for quantifying the performance of text extraction. Yet the task of extracting text from a file is comparable to the task of applying optical character recognition (OCR) to a document image. Furthermore, character error rate (CER) is well established for scoring OCR performance. So, we decided to apply CER to scoring the quality of PDF-extracted text.

Fortunately, there are open source tools available for scoring CER and analyzing OCR output. We used the University of Nevada Las Vegas (UNLV) Information Science Research Institute (ISRI) Analytic Tools for OCR Evaluation v.5.1 and the UNLV ISRI OCR Frontiers Toolkit v.1.0 to score the character error (UNLV, 1996; Rice & Nartker, 1996; UNLV, 1999; Bagdanov, Rice, & Nartker, 1999). These include tools for dealing with Unicode data—the format of our Arabic and Persian original plain text, and Arabic and Persian PDF-extracted text.[10,11] The Accuracy tool generates the Accuracy score that we used to rank the PDF generation methods, and generates a number of other statistics that are useful for analyzing character errors. The Synctext tool provides error position information.

The UNLV tools allow us to compare some PDF-extracted text output—the *hypothesis*—against some original plain text document—the *reference*. UNLV's definition of Accuracy is as follows: given a reference and a hypothesis, $Accuracy = \frac{Characters - Errors}{Characters}$, where *Characters* is the total

[10] Ideally, we would also want to score word error rate (or token error rate), but the UNLV tool documentation states that the word accuracy tools define a word as a sequence of one or more ASCII or Latin1 letters.
[11] These UNLV tools convert Unicode to and from a format called *Extended ASCII*.

number of characters in the reference, and *Errors* is the total number of edit operations needed to correct the hypothesis and make it identical to the reference (i.e., insertions, deletions and substitutions).

The first step of the pipeline was to create the *reference* file for our test. For this we normalized the original plain text UTF-8 document, using Unicode Normal Form Compatibility Decomposition (NFKD). This provided a canonical Unicode UTF-8 *reference* file for the scoring step. Hence, our *reference* file is a *normalized reference*.

The plan was to compare all PDF-extracted text output files—our *hypothesis* files—against the *reference*. The PDF extractor generated UTF-8, but, of course, we could not guarantee that each hypothesis file would be NFKD normalized. So, we applied two treatments to each hypothesis file. In the first case, we applied Unicode NFKD normalization—a *normalized* hypothesis file. In the second, we applied no normalization at all—an *unnormalized* hypothesis file. We expected that *normalized* hypotheses would provide a closer match to the reference, hence giving each method a better chance for a higher score. If normalized and unnormalized scores differed, we planned to be fair and report the higher score.

In all, we scored 32 combinations of language, PDF generation method, and hypothesis treatment.

Further manipulations (i.e., conversions) were necessary to match our UTF-8 data to the UNLV tools, which work with UTF-16. We used iconv v.1.9, distributed with Gnu libiconv 1.9.2, to perform such conversions.

We based the ranking of PDF generation methods on the Accuracy scores in the UNLV Accuracy reports.

2.8. Data Analysis

The data analysis step involved a variety of tools. The UNLV tool reports provided insights into insertions, deletions, and substitutions. But examining the PDF Language in each PDF file was vital for locating at which point in the pipeline each error appeared. Acrobat's Preflight tool provided invaluable insight into the ordering of text objects in each PDF prior to wading into the PDF code. A custom profile listed all text objects. We used Pdftk v.1.41 to decompress FlateDecode streams in the PDFs (Pdftk, 2010). Manual analysis of the PDF code and cross-referencing to UNLV tool reports was time consuming and tedious. Automating the decoding of font code strings helped to a degree. For this, we developed a rudimentary, Perl-based font encoding to UTF-8 converter, which allowed us to read Arabic and Persian font code strings (i.e., Arabic script).

3. Results and Discussion

In this section we present the PDF generation method Accuracy scores. Then, we explain PDF generation classes and their relation to the Accuracy scores. Each section of results is followed by discussion.

3.1 Results: Scores and Ranking

Table 3 shows a summary of the results from UNLV Accuracy reports. The Filename column shows how the data is grouped. The *Filename* identifies the hypothesis treatment, the language, and the PDF generation method. *Nrm*, or *Unm*, indicates the hypothesis treatment. *Ara*, or *Per*, indicates the language. And the index number identifies the PDF generation method, as listed at the beginning of section 2.5.

The column data associated with each Filename needs some explanation, which comes from Rice, et al. (1996) and Bagdanov, et al., (1999). It begins with three columns for overall scores. *Overall Characters* is the total number of characters in the reference. *Overall Errors* is the sum of insertions, deletions, and substitutions required to make the hypothesis identical to the reference. *Overall Accuracy* is the *Overall Characters* minus the *Overall Errors*, divided by the *Overall Characters*. This latter score is the score we intended to use for ranking the methods.

The subsequent columns quantify the amount of post-editing work related to categories of reference characters. Each character category is specified by the category name and represented by three columns: *Count*, *Missed*, and *Percentage Right*. Each column labeled *Count* is the total number of characters appearing in the reference. Each column labeled *Missed* is the total number of insertions plus substitutions required to restore the reference characters. Each third column is $Percent\ Right = \frac{Count - Missed}{Count}$. Character categories labeled *ASCII* are more or less understood, but we could find no documentation from UNLV on the specific code point ranges, or sets, of other categories: *Special Symbols*, *Basic Arabic*, *Arabic Extended*, and *Punctuation*. Nevertheless, we found them somewhat helpful.

The final three columns reflect the post-editing work for all character categories: the *Total* Count, Missed, and Percentage Right.

Table 3(a). Accuracy report statistics by PDF generation method, language, and treatment

Filename	Overall Characters	Overall Errors (Insertions +Deletions +Sub-stitutions)	Overall Accuracy	Count ASCII Spacing Characters	Missed ASCII Spacing Characters	Percentage Right ASCII Spacing Characters	Count ASCII Special Symbols	Missed ASCII Special Symbols	Percentage Right ASCII Special Symbols
Rpt-NrmAra01.txt	936	44	95.3	150	10	93.33	7	1	85.71
Rpt-NrmAra02.txt	936	44	95.3	150	10	93.33	7	1	85.71
Rpt-NrmAra03.txt	936	44	95.3	150	10	93.33	7	1	85.71
Rpt-NrmAra04.txt	936	44	95.3	150	10	93.33	7	1	85.71
Rpt-NrmAra05.txt	936	599	36	150	10	93.33	7	1	85.71
Rpt-NrmAra06.txt	936	44	95.3	150	10	93.33	7	1	85.71
Rpt-NrmAra07.txt	936	44	95.3	150	10	93.33	7	1	85.71
Rpt-NrmAra08.txt	936	44	95.3	150	10	93.33	7	1	85.71
Rpt-NrmPer01.txt	957	57	94.04	173	11	93.64	5	0	100

Filename	Overall Characters	Overall Errors (Insertions +Deletions +Sub-stitutions)	Overall Accuracy	Count ASCII Spacing Characters	Missed ASCII Spacing Characters	Percentage Right ASCII Spacing Characters	Count ASCII Special Symbols	Missed ASCII Special Symbols	Percentage Right ASCII Special Symbols
Rpt-NrmPer02.txt	957	57	94.04	173	11	93.64	5	0	100
Rpt-NrmPer03.txt	957	62	93.52	173	11	93.64	5	0	100
Rpt-NrmPer04.txt	957	57	94.04	173	11	93.64	5	0	100
Rpt-NrmPer05.txt	957	618	35.42	173	11	93.64	5	2	60
Rpt-NrmPer06.txt	957	57	94.04	173	11	93.64	5	0	100
Rpt-NrmPer07.txt	957	62	93.52	173	11	93.64	5	0	100
Rpt-NrmPer08.txt	957	57	94.04	173	11	93.64	5	0	100
Rpt-UnmAra01.txt	936	44	95.3	150	10	93.33	7	1	85.71
Rpt-UnmAra02.txt	936	44	95.3	150	10	93.33	7	1	85.71
Rpt-UnmAra03.txt	936	38	95.94	150	10	93.33	7	1	85.71
Rpt-UnmAra04.txt	936	44	95.3	150	10	93.33	7	1	85.71
Rpt-UnmAra05.txt	936	543	41.99	150	10	93.33	7	1	85.71
Rpt-UnmAra06.txt	936	44	95.3	150	10	93.33	7	1	85.71
Rpt-UnmAra07.txt	936	38	95.94	150	10	93.33	7	1	85.71
Rpt-UnmAra08.txt	936	44	95.3	150	10	93.33	7	1	85.71
Rpt-UnmPer01.txt	957	57	94.04	173	11	93.64	5	0	100
Rpt-UnmPer02.txt	957	57	94.04	173	11	93.64	5	0	100
Rpt-UnmPer03.txt	957	57	94.04	173	11	93.64	5	0	100
Rpt-UnmPer04.txt	957	57	94.04	173	11	93.64	5	0	100
Rpt-UnmPer05.txt	957	569	40.54	173	11	93.64	5	2	60
Rpt-UnmPer06.txt	957	57	94.04	173	11	93.64	5	0	100
Rpt-UnmPer07.txt	957	57	94.04	173	11	93.64	5	0	100
Rpt-UnmPer08.txt	957	57	94.04	173	11	93.64	5	0	100

Table 3(b). Accuracy report statistics by PDF generation method, language, and treatment (Continued)

Filename	Count ASCII Digits	Missed ASCII Digits	Percentage Right ASCII Digits	Count Basic Arabic	Missed Basic Arabic	Percentage Right Basic Arabic	Count Arabic Extended	Missed Arabic Extended	Percentage Right Arabic Extended
Rpt-NrmAra01.txt	6	0	100	701	1	99.86	70	27	61.43
Rpt-NrmAra02.txt	6	0	100	701	1	99.86	70	27	61.43
Rpt-NrmAra03.txt	6	0	100	701	6	99.14	70	4	94.29
Rpt-NrmAra04.txt	6	0	100	701	1	99.86	70	27	61.43
Rpt-NrmAra05.txt	6	4	33.33	701	494	29.53	70	7	90
Rpt-NrmAra06.txt	6	0	100	701	1	99.86	70	27	61.43
Rpt-NrmAra07.txt	6	0	100	701	6	99.14	70	4	94.29
Rpt-NrmAra08.txt	6	0	100	701	1	99.86	70	27	61.43
Rpt-NrmPer01.txt	7	0	100	672	0	100	95	36	62.11
Rpt-NrmPer02.txt	7	0	100	672	0	100	95	36	62.11
Rpt-NrmPer03.txt	7	0	100	672	5	99.26	95	24	74.74
Rpt-NrmPer04.txt	7	0	100	672	0	100	95	36	62.11
Rpt-NrmPer05.txt	7	4	42.86	672	512	23.81	95	25	73.68
Rpt-NrmPer06.txt	7	0	100	672	0	100	95	36	62.11

Filename	Count ASCII Digits	Missed ASCII Digits	Percentage Right ASCII Digits	Count Basic Arabic	Missed Basic Arabic	Percentage Right Basic Arabic	Count Arabic Extended	Missed Arabic Extended	Percentage Right Arabic Extended
Rpt-NrmPer07.txt	7	0	100	672	5	99.26	95	24	74.74
Rpt-NrmPer08.txt	7	0	100	672	0	100	95	36	62.11
Rpt-UnmAra01.txt	6	0	100	701	1	99.86	70	27	61.43
Rpt-UnmAra02.txt	6	0	100	701	1	99.86	70	27	61.43
Rpt-UnmAra03.txt	6	0	100	701	6	99.14	70	4	94.29
Rpt-UnmAra04.txt	6	0	100	701	1	99.86	70	27	61.43
Rpt-UnmAra05.txt	6	4	33.33	701	494	29.53	70	6	91.43
Rpt-UnmAra06.txt	6	0	100	701	1	99.86	70	27	61.43
Rpt-UnmAra07.txt	6	0	100	701	6	99.14	70	4	94.29
Rpt-UnmAra08.txt	6	0	100	701	1	99.86	70	27	61.43
Rpt-UnmPer01.txt	7	0	100	672	0	100	95	36	62.11
Rpt-UnmPer02.txt	7	0	100	672	0	100	95	36	62.11
Rpt-UnmPer03.txt	7	0	100	672	5	99.26	95	24	74.74
Rpt-UnmPer04.txt	7	0	100	672	0	100	95	36	62.11
Rpt-UnmPer05.txt	7	4	42.86	672	514	23.51	95	24	74.74
Rpt-UnmPer06.txt	7	0	100	672	0	100	95	36	62.11
Rpt-UnmPer07.txt	7	0	100	672	5	99.26	95	24	74.74
Rpt-UnmPer08.txt	7	0	100	672	0	100	95	36	62.11

Table 3(c). Accuracy report statistics by PDF generation method, language, and treatment (Continued)

Filename	Count General Punctuation	Missed General Punctuation	Percentage Right General Punctuation	Count Total	Missed Total	Percentage Right Total
Rpt-NrmAra01.txt	2	2	0	936	41	95.62
Rpt-NrmAra02.txt	2	2	0	936	41	95.62
Rpt-NrmAra03.txt	2	2	0	936	23	97.54
Rpt-NrmAra04.txt	2	2	0	936	41	95.62
Rpt-NrmAra05.txt	2	2	0	936	518	44.66
Rpt-NrmAra06.txt	2	2	0	936	41	95.62
Rpt-NrmAra07.txt	2	2	0	936	23	97.54
Rpt-NrmAra08.txt	2	2	0	936	41	95.62
Rpt-NrmPer01.txt	5	5	0	957	52	94.57
Rpt-NrmPer02.txt	5	5	0	957	52	94.57
Rpt-NrmPer03.txt	5	5	0	957	45	95.3
Rpt-NrmPer04.txt	5	5	0	957	52	94.57
Rpt-NrmPer05.txt	5	5	0	957	559	41.59
Rpt-NrmPer06.txt	5	5	0	957	52	94.57
Rpt-NrmPer07.txt	5	5	0	957	45	95.3
Rpt-NrmPer08.txt	5	5	0	957	52	94.57
Rpt-UnmAra01.txt	2	2	0	936	41	95.62
Rpt-UnmAra02.txt	2	2	0	936	41	95.62
Rpt-UnmAra03.txt	2	2	0	936	23	97.54
Rpt-UnmAra04.txt	2	2	0	936	41	95.62

Filename	Count General Punctuation	Missed General Punctuation	Percentage Right General Punctuation	Count Total	Missed Total	Percentage Right Total
Rpt-UnmAra05.txt	2	2	0	936	517	44.76
Rpt-UnmAra06.txt	2	2	0	936	41	95.62
Rpt-UnmAra07.txt	2	2	0	936	23	97.54
Rpt-UnmAra08.txt	2	2	0	936	41	95.62
Rpt-UnmPer01.txt	5	5	0	957	52	94.57
Rpt-UnmPer02.txt	5	5	0	957	52	94.57
Rpt-UnmPer03.txt	5	5	0	957	45	95.3
Rpt-UnmPer04.txt	5	5	0	957	52	94.57
Rpt-UnmPer05.txt	5	5	0	957	560	41.48
Rpt-UnmPer06.txt	5	5	0	957	52	94.57
Rpt-UnmPer07.txt	5	5	0	957	45	95.3
Rpt-UnmPer08.txt	5	5	0	957	52	94.57

This data allowed us to achieve the ranking. We observed that there are many redundant values in this table, which led us to think that there was something common among the files. Analysis of PDF file properties revealed that these eight PDF generation methods are in fact related to three *PDF generation classes*—Acrobat Distiller/PScript5.dll, PDFMaker, and Microsoft Word.[12] Hence, our 8 methods invoked only 3 PDF generation classes. The Unix diff command confirmed that the hypothesis files for a given class, language, and hypothesis treatment, are identical. Table 4 lists the relationship between class and method, the best Accuracy scores, and the rank based on Accuracy. Acrobat Distiller/PScript5.dll and PDFMaker received comparably high scores and share rank 1 (96% and 95% respectively for Arabic; 94% for Persian). Microsoft Word ranked second due to very low Accuracy scores.

Table 4. PDF generation classes by rank

PDF Generation Class	PDF Generation Method	Overall Accuracy: Arabic*	Overall Accuracy: Persian*	Rank
Acrobat Distiller/PScript5.dll	3, 7	96%	94%	1
PDFMaker	1, 2, 4, 6, 8	95%	94%	1
Microsoft Word	5	42%	41%	2

** These are unnormalized scores. See the following section for discussion.*

3.2. Discussion: Scores and Ranking

We need to give a word of caution about correlating these Accuracy scores with Accuracy scores in other reports. These Accuracy scores do not reflect the type of Accuracy that we would see on real documents. Accuracy typically describes performance on a real-world human language document—and we did not use real-world documents. Rather we used systematically fabricated documents intended for comprehensive glyph coverage.

Many of the normalized Accuracy scores are equal to or lower than the unnormalized scores. This is the reverse of what we expected. We examined the Accuracy reports for a few of the

[12] The class names here are from the Application and/or PDF-Producer property values revealed by Windows XP Windows Explorer via right clicking each PDF and selecting Properties (from the context menu).

language and method combinations with the largest normalized and unnormalized score differences. Although most of the errors are substitution errors, the reports for the normalized files show an inordinately high count of deletion errors. According to the Unicode standard, there are many single canonical characters that are *Unicode canonically equivalent* to double canonical characters. Although, technically, these are not errors, such single characters in the hypothesis and double characters in the reference would result in a high count of deletion errors. For example, each Acrobat Distiller/PScript5.dll unnormalized hypothesis included ALEF MADDAH (آ U+0622), whereas each corresponding normalized hypothesis included its Unicode canonical equivalent: ARABIC LETTER ALEF and ARABIC MADDAH ABOVE (ا U+0627 and ٓ U+0653). This resulted in additional reported deletion errors for the normalized hypotheses: 6 for Arabic (a 0.52% Accuracy difference), and 5 for Persian (a 0.64% Accuracy difference).

Each PDFMaker normalized score was identical to its unnormalized score. Each Acrobat Distiller/PScript5.dll normalized score was identical to its unnormalized score, except in the case noted above.[13]

Comparably high Acrobat Distiller/PScript5.dll and PDFMaker scores do not necessarily indicate that these are good PDF generation classes from the perspective of human language technology. At best there was a 4% to 6% character error rate. Such error rates may correspond to an even higher percentage of word/token errors. That degree of word/token error rate is problematic for human language technology applications. Only an analysis of specific errors would tell the complete story. So, our data analysis turned to correlating specific character errors shown in the UNLV reports with the PDF code in actual files.

3.3. Results: Acrobat Distiller/PScript5.dll PDF Generation Class Errors

We first address the Acrobat Distiller/PScript5.dll PDF generation class. Table 5 shows the specific errors. The first column provides the error description. The next column shows the error type: *Character* or *Entity*. Each error labeled *Character* error type is an insertion, substitution, and/or deletion of a specific Unicode character. The subsequent two columns list the number of times the character error occurred in a normalized Arabic hypothesis and normalized Persian hypothesis. Some frequencies are accompanied by the number of tokens affected. We observed errors with alphabetic characters, numeric characters, spacing characters, and a symbol. Most character errors are footnoted with an asterisk to indicate that the count represents all occurrences of the reference character. We use multiple rows to describe the total set of space insertion errors (14 for Arabic; 13 for Persian).

Each error listed as an *Entity* error type reflects a mishandling of either a date or measurement entity. These were due to one or more character errors, or a reversal of the character order. We observed a reversal in the order of the slash delimited date in the Arabic hypotheses, but the hyphen delimited date, and the Persian storage order YYYY/MM/DD date, were handled correctly in all cases.

The final column reports whether or not the error appears in the PDF Language code of the PDF files.

[13] Identical sets of scores also indicated identical files.

Table 5. Acrobat Distiller/PScript5.dll PDF generation class errors.

Error Description	Error Type	Frequency in Normalized Arabic Hypothesis	Frequency in Normalized Persian Hypothesis	Error observed in PDF file
ARABIC LETTER KAF (ك U+0643), where it renders as initial form, substituted by ARABIC LETTER ALEF (ا U+0627) and ARABIC MADDAH ABOVE (ٓ U+0653) (i.e., آ Alef Maddah)	Character	6*	5*	Yes
ARABIC LETTER FARSI YEH (ی U+06CC), where rendered as initial or medial form, substituted with ARABIC LETTER YEH (ي U+064A)	Character	N/A	3*	Yes
Eastern Arabic-Indic Digits ۰ (U+06F0), ۱ (U+06F1), ۲ (U+06F2), ۳ (U+06F3), ۷ (U+06F7), ۸ (U+06F8), and ۹ (U+06F9), substituted with Arabic-Indic Digits ٠ (U+0660), ١ (U+0661), ٢ (U+0662), ٣ (U+0663), ٧ (U+0667), ٨ (U+0668), and ٩ (U+0669), respectively	Character	N/A	21*	Yes
PERIOD (. U+002E) substituted with the ARABIC DECIMAL SEPARATOR (٫ U+066B)	Character	1*	N/A	Yes
SPACE (U+0020), the token delimiter, substituted with a newline (i.e., line break)	Character	10	10	N/A[14]
ZERO WIDTH NON-JOINER (U+200C) deleted and collocated with an inserted SPACE	Character	2* (1 token)	5* (5 tokens)	Yes
In a given token ending with ARABIC FATHATAN (ً U+064B), one or more correct characters are each followed by an inserted SPACE	Character	6 (4 tokens)	1 (1 token)	No
In a given token ending with ARABIC HAMZA ABOVE (ٔ U+0654), one or more correct characters are each followed by an inserted SPACE	Character	6 (2 tokens)	7 (3 tokens)	No
DD/MM/YYYY date entity (storage order) substituted with YYYY/MM/DD date (storage order)	Entity	1	N/A	No
Measurement entity error due to the PERIOD error mentioned above	Entity	1	N/A	Yes
Date entity error due to the Eastern Arabic-Indic Digits error mentioned above	Entity	N/A	3	Yes
Measurement entity error due to the ARABIC LETTER FARSI YEH error mentioned above	Entity	N/A	1	Yes

* This accounts for all of the occurrences of the reference character as described.

3.4. Discussion: Acrobat Distiller/PScript5.dll PDF Generation Class Errors

In this section, we discuss high impact errors, such as the ARABIC LETTER KAF substitutions, the ZERO WIDTH NON-JOINER deletions, the date entity errors, and the spurious space insertions. However, we begin by discussing the benign errors and the errors of lesser significance.

One benign error is the ARABIC LETTER FARSI YEH substitution, a common, accepted substitution for human keyboarding of Persian. As such, it would have to be accommodated in an HLT application.

[14] It could be argued that the error *was* observed in the PDF because the PDF generation method discarded the fact that there were no newlines in the original text.

Other errors would likely cause problems for human language technology applications. Eastern Arabic-Indic Digit substitutions are acceptable in Persian, but a PDF generation method should consistently substitute all digits in a numeric expression. Rather, we observed consistent substitution of only 7 of the 10 digits. These are the seven digits for which Arabic-Indic and Eastern Arabic-Indic digits have the same glyph. This may confuse a natural language processing application.

Spaces substituted with newlines would not necessarily affect search engine indexing because each token gets individually indexed anyway. However, such spurious newlines break up complete sentences; and tools such as machine translation engines, text analytics (e.g., named entity extraction, information extraction), and computer-assisted translation tools may be counting on newlines to indicate sentence breaks.

The other insertions, deletions, and substitutions that are listed here are significant errors that would confuse most HLT applications—for example, rendering unrecognizable every Arabic word that begins with ARABIC LETTER KAF. Each of these character errors causes at least one token error, and token errors degrade the performance of most human language technologies (machine translation, information extraction, etc.) Character errors also render unreadable important entities, such as the dates and measurements included in our test documents. Three of the entity errors listed are due to character errors.

Overall, the deleted ZERO WIDTH NON-JOINERs and substituted digits have a greater impact on Persian than Arabic because they are more prevalent in Persian.

The reversal in the order of the date components is of particular interest because, for many use cases, accurate dates are vitally important. Consider the date 01/02/03 after it is changed to 03/02/01. This latter date is an entirely different day than the former, but might be interpreted by a human as an authoritative date. The date in our Arabic document is storage order DD/MM/YYYY format; storage order is the important factor here. The date entity text is in the PDF, but has been split apart into 5 strings. Still, it is a valid PDF. However, the extractor was unable to recover the original date entity in the correct order. This appears to be a specific case in which our test configuration would generate an error.

We were surprised to find that most of the errors were caused by the PDF generation software. That is, the errors were introduced at the time the PDF was generated and not at the time the text was extracted from the PDF. The PDF generation software caused the ARABIC LETTER KAF substitutions, the ARABIC LETTER FARSI YEH substitutions, the Eastern Arabic-Indic Digit substitutions, the PERIOD substitutions, and the ZERO WIDTH NON-JOINER deletions. An argument could be made that the PDF generation software caused the newline insertions because the PDF generation discarded the fact that the original electronic text had no newlines. Due to the character errors, the PDF generation software also caused most of the entity errors.

The text extraction software caused the space insertions in tokens ending with Fathatan and Hamza. This appears to be due to the text extractor not properly dealing with a complex construction in the PDF Language code. This only occurred where Acrobat Distiller/PScript5.dll encoded a specific combination of two text-related operations.

To understand the PDF Language that caused trouble for the text extraction software, it is necessary to first understand a little about the PDF *TJ* and *Tc operators*. The *TJ operator* is a text showing operator. It renders a font code string on the page. The *Tc operator* sets the default character spacing added after each glyph is rendered. Normally, as the TJ operator renders each character, the current position moves to the right *the width of the glyph plus the Tc character spacing*. But, the TJ operator allows a horizontal kerning value after each font code to affect a minute adjustment to the left. An example of kerning is the horizontal adjustment to change *Text* to *Text*. The latter shows a smaller separation between the T and e. For further information on such typographic concepts, we refer the reader elsewhere for overviews of the topic (Bringhurst, 2004; Lupton, 2010). If the TJ operator specifies no kerning value after a font code, then no horizontal adjustment is made.

An effective PDF text extractor must not only understand the nuances of the TJ and Tc operators, it must understand *all* of the nuances of the PDF Language. In cases involving complex positioning of glyphs on the page, the text extractor must apply heuristics in order to determine whether such positioning indicates a space or a newline—and how many spaces or newlines (Rosenthol, 2010). Where such whitespace is absent from the PDF, the extractor must reverse engineer, and generate, the missing electronic text. In this latter case, the term "PDF text *extractor*" is really a misnomer—it is an extractor *and* generator.

We now turn to the specific case that confused the text extractor in our tests—causing the insertion of spurious spaces. Here we talk about the font codes of the TJ operator as glyphs. In this case, the character spacing was large and negative, such that, by default, each glyph would render directly on top of the others (i.e., render a glyph, then move left to the beginning of the glyph we just rendered—which is the opposite of the typical left-to-right movement). This works well to render the diacritic over the final letter of a token, but a large negative kerning value is necessary to successively move rightward and render each remaining glyph (i.e., use a negative kerning value to move right an entire character width—which is the opposite of the typical leftward movement for kerning.)[15] Figure 2 shows a relevant snippet of the PDF code.

```
/TT2 1 Tf
2.2322 0 TD
0 Tc
( )Tj
/TT1 1 Tf
.2772 0 TD
-.207 Tc
[<053903f1>-205.5<03ae>-208.9<03d8>-210.2<03e4>-206.2<03df>-206<038d>]TJ
```

Figure 2. PDF code snippet for a token ending in Hamza.

The cases where Acrobat Distiller/PScript5.dll deleted the ZERO WIDTH NON-JOINER characters are similar. We analyzed PDF code snippets in each of these cases, but we could not determine whether the inserted space was coded in the PDF or merely an artifact of the extractor. However, it was clear that, at the position of each deleted ZERO WIDTH NON-JOINER, the PDF generator divided the token into parts and put the parts into two or more text showing operators.

[15] In all of these cases, Tc was approximately -.200, and the TJ operator kerning values were approximately -200.

3.5. Results: PDFMaker PDF Generation Class Errors

We observed that many of the PDFMaker errors were the same errors that we observed with Acrobat Distiller/PScript5.dll: ARABIC LETTER FARSI YEH substitution, Eastern Arabic Indic Digit substitution, PERIOD substitution, space substitution, ZERO WIDTH NON-JOINER deletion, and the entity errors. We observed spurious space insertions collocated with the ZERO WIDTH NON-JOINER deletions (2 for Arabic; 5 for Persian). However, we did not see the spurious space insertions in tokens ending with Fathatan or Hamza.

New for this PDF generation class was the ARABIC HAMZA ABOVE substituted with ARABIC LETTER HIGH HAMZA.

Table 4 shows each error. Again, most character errors are footnoted with an asterisk to indicate that the count represents all occurrences.

Table 4. PDFMaker PDF generation class errors.

Error Description	Error Type	Frequency in Normalized Arabic Hypothesis	Frequency in Normalized Persian Hypothesis	Error observed in PDF file
ARABIC HAMZA ABOVE (ٔ U+0654) substituted with ARABIC LETTER HIGH HAMZA (ٴ U+0674 “Kazakh”)	Character	23*	12*	Yes
ARABIC LETTER FARSI YEH (ی U+06CC), where rendered as initial or medial form, substituted with ARABIC LETTER YEH (ي U+064A)	Character	N/A	3*	Yes
Eastern Arabic-Indic Digits ۰ (U+06F0), ۱ (U+06F1), ۲ (U+06F2), ۳ (U+06F3), ۷ (U+06F7), ۸ (U+06F8), and ۹ (U+06F9), substituted with Arabic-Indic Digits ٠ (U+0660), ١ (U+0661), ٢ (U+0662), ٣ (U+0663), ٧ (U+0667), ٨ (U+0668), and ٩ (U+0669), respectively	Character	N/A	21*	Yes
PERIOD (. U+002E) substituted with the ARABIC DECIMAL SEPARATOR (٫ U+066B)	Character	1*	N/A	Yes
SPACE (U+0020), the token delimiter, substituted with a newline (i.e., line break)	Character	10	10	N/A[16]
ZERO WIDTH NON-JOINER (U+200C) deleted and collocated with an inserted space (U+0020)	Character	2* (1 token)	5* (5 tokens)	Yes
DD/MM/YYYY date entity (storage order) substituted with YYYY/MM/DD date (storage order)	Entity	1	N/A	No
Measurement entity error due to the PERIOD error mentioned above	Entity	1	N/A	Yes
Date entity error due to the Eastern Arabic-Indic Digits error mentioned above	Entity	N/A	3	Yes
Measurement entity error due to the ARABIC LETTER FARSI YEH error mentioned above	Entity	N/A	1	Yes

* This accounts for all of the occurrences of the reference character as described.

[16] It could be argued that the error *was* observed in the PDF because the PDF generation method discarded the fact that there were no newlines in the original text.

3.6. Discussion: PDFMaker PDF Generation Class Errors

All of the discussion above about Acrobat Distiller/PScript5.dll errors applies here as well where the errors were the same. HLT applications simply will not fare well with the word errors introduced by spurious newlines, deleted non-joiners, spurious spaces, character substitutions, and entity errors.

Regarding the ARABIC LETTER HIGH HAMZA, the Unicode standard for this character is noted “Kazakh”. Hence, it is likely to be unexpected by an Arabic or Persian human language technology application and cause problems.

Again we observed the error in the slash-delimited date entity of the Arabic document. Each normalized Arabic hypothesis file had the order of the date components reversed. As with Acrobat Distiller/PScript5.dll, the date entity text is in the PDF, but the five date components have been split apart. Because the PDF is valid according to the specification, we would expect the extractor to properly parse the PDF, but it was unable. Therefore, it appears that this is an entity case for which we would expect errors.

We determined that PDFMaker not only deleted the ZERO WIDTH NON-JOINERs, but it clearly inserted the spurious spaces. Figure 3 shows a PDF code snippet where the PDF generator deleted the ZERO WIDTH NON-JOINER, separated the token into parts, put the parts into one or more text showing operators, and inserted a space. The gray highlighting denotes the each part of the single token, and the inserted space.

```
7.633 -1.15 Td
[<03E6>1<03E403A0>-3<03E7038D000303EA03CC039F>1<038D03AE>-3<03E30003>-
4<039D03AD>-3<038E03A7>1<0003039A03A4>1<0391000303AE>-3<039C03DB038D03AA>-
1<03A3>1<000303AE>-3<039B>-4<0539>206<03EE03E30003039903A9>-
1<038D03EE03A3>1<0003>-4<039603E703ED038E03CC03E3000303F1038E>-4<03EC039803F4>-
4<03DF038E03CC03D3>1<000303AA>-1<03F403F30539>206<038E0397>]TJ
/TT0 1 Tf
0 Tc 0 Tw T*
( )Tj
/C2_0 1 Tf
-0.001 Tc 0.001 Tw -8.884 0 Td
[<0003>-4<03EA032A>1<03E7038E03E80329>1<0003>-4<031F038E0329>1<0003032703AD>-
3<038E03D7>1<0003>-4<039E>1<03E8032100303F1038E03EB>]TJ
```

Figure 3. PDF code snippet for a token with a substituted ZERO WIDTH NON-JOINER.

3.7 Results: Microsoft Word PDF Generation Class Errors

The summary scores sufficiently describe the data related to the Microsoft Word PDF generation class errors. The high error rate made it unreasonable for us to pursue a more detailed analysis of the related files.

4. Conclusion

The current study ranked the PDF generation methods used for converting Arabic and Persian digital documents to PDF. It identified that the eight methods are merely eight paths to three PDF generation classes: Acrobat Distiller/PScript5.dll, PDFMaker, and Microsoft Word. The study showed that the scores of the former two (96% and 95% at best) are superior to that of the latter (42% at best). Table 5 shows a summary of the ranking. This study revealed that there is a significant loss of reliable electronic text with all of the methods, and that this poses significant problems for human language technology applications such as search engines, machine translation engines, computer-assisted translation tools, named entity recognizers, and information extractors.

In the study, most of the spurious newlines, spurious spaces in tokens, spurious character substitutions, and entity errors were due to the PDF generation method, rather than the PDF text extractor.

Table 5. Ranked PDF Generation Methods

Rank	PDF Generation Method	PDF Generation Class	Best Overall Accuracy: Arabic	Best Overall Accuracy: Persian
1	(3) Open the .DOCX file in Word; print it to the Adobe printer (i.e., virtual printer) (7) Ensure the default printer is set to the Adobe printer (i.e., virtual printer); right-click the .DOCX file in Windows Explorer; click Print	Acrobat Distiller /PScript5.dll	96%	94%
1	(1) Open the .DOCX file in Acrobat; click File > Save As; click Save (to PDF) (2) Open Acrobat; click File > Create PDF > From File; specify the .DOCX file; click Open; click Save (to PDF) (4) Open the .DOCX file in Word; click the Microsoft Office Button; click Save As > Adobe PDF (6) Open the .DOCX file in Word; click the Acrobat tab; click Create PDF (8) Right-click the .DOCX file in Windows Explorer; click Convert To Adobe PDF; click Save	PDFMaker	95%	94%
2	(5) Open the .DOCX file in Word; click the Microsoft Office Button; click Save As > PDF or XPS; click Publish	Microsoft Word	42%	41%

These observations were made using a common, best-practice setup in order to generate PDFs and extract text. We believe that Adobe Acrobat represents the best-of-breed. It remains to be seen what errors would result from other PDF generation configurations. For example, a follow on study would be needed to determine what errors would be realized from using PDF Creator rather than Acrobat, Microsoft Outlook rather than Microsoft Word, or Times New Roman font rather than Arial font.

The bottom line here is that attempting to store reliable electronic text for data exchange with typographic commands on a Cartesian coordinate system is fraught with complications.

Acknowledgments

We wish to thank Marwan Awad and Karine Megerdoomian, without whom this study would not have been possible. These collaborators contributed their language expertise to the building and validating of the token lists. The token lists, in turn, allowed us to build the test documents and complete the study.

We would like to acknowledge the support of the HighView team at CACI for assisting with the setup and configuration of the PDF extraction process.

References

Adobe (2008). Document management — Portable document format — Part 1: PDF 1.7. Retrieved June 16, 2010 from http://www.adobe.com/devnet/acrobat/pdfs/PDF32000_2008.pdf

Apache. (2010). Apache PDFBox 1.3.1 [Computer software]. Retrieved December 10, 2010 from http://pdfbox.apache.org/download.html

Bagdanov, A., Rice, S., & Nartker, T. (1999). The OCR frontiers toolkit, version 1.0. Information Science Research Institute. University of Nevada, Las Vegas. Retrieved April 23, 2009 from http://www.isri.unlv.edu/downloads/ftk-1.0.tgz[17]

Bringhurst, R. (2004). *The elements of typographic style*. Point Roberts, WA: Hartley and Marks Publishers.

Carrier, B. (2009). Extracting searchable text from Arabic PDFs. Basis Technology. Retrieved September 15, 2010 from http://www.basistech.com/knowledge-center/forensics/extracting-text-from-Arabic-PDF.pdf

Herceg, P. M., & Ball, C. N. (2010). Reliable electronic text: The elusive prerequisite for a host of human language technologies. Technical Report MTR100302. McLean, VA: The MITRE Corporation.

King, J. (2008, July 29). Text content in PDF files. Article posted to http://blogs.adobe.com/insidepdf/2008/07/text_content_in_pdf_files.html (Adobe Blogs).

Lupton, E. (2010). *Thinking with type*. New York: Princeton Architectural Press.

Powley, B., Dale, R., & Anisimoff, I. (2009). Enriching a document collection by integrating information extraction and PDF annotation. In Proceedings of the International Society for Optical Engineering (SPIE) Vol. 7247, Document Recognition and Retrieval XVI. Retrieved October 20, 2010 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.159.1718&rep=rep1&type=pdf

[17] This resource is no longer available at the specified URL. Contact Thomas Nartker at tnartker@cs.unlv.edu.

Pdftk. (2010). Pdftk: The PDF toolkit (v.1.41) [Computer Software]. Retrieved July 7, 2010 from http://www.pdfhacks.com/pdftk/

Rice, S., & Nartker, T. (1996). The ISRI analytic tools for OCR evaluation, version 5.1. Technical Report TR-96-02. Information Science Research Institute. University of Nevada, Las Vegas. Retrieved April 23, 2009 from http://www.isri.unlv.edu/downloads/at-user-guide.pdf[17]

Rosenthol, L. (2010, September 14). Determining word boundaries when no space exists in text. Retrieved January 11, 2011 from http://forums.adobe.com/thread/720790?tstart=0

Unicode Consortium. (2011). Ligatures, digraphs, presentation forms vs. plain text. Retrieved February 9, 2011 from http://unicode.org/faq/ligature_digraph.html

UNLV. (1996). The ISRI analytic tools for OCR evaluation, version 5.1 [Computer software]. Retrieved April 23, 2009 from http://www.isri.unlv.edu/ISRI/OCRtk[17]

UNLV. (1999). The ISRI OCR frontiers toolkit, version 1.0 [Computer software]. Retrieved April 23, 2009 from http://www.isri.unlv.edu/downloads/ftk-1.0.tgz[17]